Capítulo DEZOITO

Streams Paralelos

Objetivos do Exame

Usar Streams paralelos incluindo redução, decomposição, processos de junção, pipelines e desempenho.

O que é um Stream Paralelo?

Até agora, todos os exemplos desta seção usaram streams sequenciais, onde cada elemento é processado um por um.

Em contraste, streams paralelos dividem o stream em várias partes. Cada parte é processada por uma thread diferente ao mesmo tempo (em paralelo).

Nos bastidores, streams paralelos usam o **Fork/Join Framework** (que revisaremos em um capítulo posterior).

Isso significa que, por padrão, o número de threads disponíveis para processar streams paralelos é igual ao número de núcleos disponíveis do processador da sua máquina.

A vantagem de usar streams paralelos sobre o Fork/Join Framework é que eles são mais fáceis de usar.

Para criar um stream paralelo, basta usar o método parallel():

```java
Stream<String> parallelStream =
              Stream.of("a","b","c").parallel();
```

Para criar um stream paralelo a partir de uma Collection, use o método parallelStream():

```java
List<String> list = Arrays.asList("a","b","c");
Stream<String> parStream = list.parallelStream();
```

Como Funcionam os Streams Paralelos?

Vamos começar com o exemplo mais simples:

```java
Stream.of("a","b","c","d","e")
      .forEach(System.out::print);
```

Imprimir uma lista de elementos com um stream sequencial exibirá o resultado esperado:

```nginx
abcde
```

Contudo, ao usar um stream paralelo:

```java
Stream.of("a","b","c","d","e")
      .parallel()
      .forEach(System.out::print);
```

A saída pode ser diferente a cada execução:

```cpp
cbade // Uma execução
cebad // Outra execução
cbdea // Outra execução ainda
```

A razão é a decomposição do stream em partes e seu processamento por threads diferentes, como mencionamos antes.

Portanto, streams paralelos são mais apropriados para operações onde a ordem de processamento não importa e que não precisam manter estado (elas são sem estado e independentes).

Um exemplo para ver essa diferença é o uso de findFirst() vs. findAny().

No capítulo anterior, mencionamos que o método findFirst() retorna o primeiro elemento de um stream. Mas ao usar streams paralelos e ele ser decomposto em múltiplas partes, este método precisa "saber" qual elemento é o primeiro:

```java
long start = System.nanoTime();
String first = Stream.of("a","b","c","d","e")
    .parallel().findFirst().get();
long duration = (System.nanoTime() - start) / 1000000;
System.out.println(first + " found in " + duration + " milliseconds");
```

Saída:

```css
a found in 2.436155 milliseconds
```

Por isso, se a ordem não importa, é melhor usar findAny() com streams paralelos:

```java
long start = System.nanoTime();
String any = Stream.of("a","b","c","d","e")
    .parallel().findAny().get();
long duration = (System.nanoTime() - start) / 1000000;
System.out.println(any + " found in " + duration + " milliseconds");
```

Saída:

```r
c found in 0.063169 milliseconds
```

Como um stream paralelo é processado, bem, em paralelo, é razoável acreditar que ele será processado mais rápido que um stream sequencial. Mas como você pode ver com findFirst(), isso nem sempre é o caso.

Operações com Estado

Operações com estado, como:

```java
Stream<T> distinct()
Stream<T> sorted()
Stream<T> sorted(Comparator<? super T> comparator)
Stream<T> limit(long maxSize)
Stream<T> skip(long n)
```

Incorporam estado de elementos processados anteriormente e podem precisar percorrer o stream inteiro para produzir um resultado, então não são adequadas para streams paralelos.

Aliás, você pode transformar um stream paralelo em um sequencial com o método sequential():

```java
stream
   .parallel()
      .filter(..)
         .sequential()
            .forEach(...);
```

Verifique se um stream é paralelo com isParallel():

```java
stream.parallel().isParallel(); // true
```

E transforme um stream ordenado em não ordenado (ou garanta que o stream seja não ordenado) com unordered():

```java
stream
   .parallel()
      .unordered()
         .collect(...);
```

Mas não acredite que, ao executar primeiro as operações com estado e depois transformar o stream em paralelo, o desempenho será melhor em todos os casos, ou pior, toda a operação pode ser executada em paralelo, como no exemplo a seguir:

```java
double start = System.nanoTime();
Stream.of("b","d","a","c","e")
    .sorted()
    .filter(s -> {
        System.out.println("Filter:" + s);
        return !"d".equals(s);
    })
    .parallel()
    .map(s -> {
        System.out.println("Map:" + s);
        return s += s;
    })
    .forEach(System.out::println);
double duration = (System.nanoTime() - start) / 1_000_000;
System.out.println(duration + " milliseconds");
```

Pode-se pensar que o stream é ordenado e filtrado sequencialmente, mas a saída mostra outra coisa:

```less
Filter:c
Map:c
cc
Filter:a
Map:a
aa
Filter:b
Map:b
bb
Filter:d
Filter:e
Map:e
ee
79.470779 milliseconds
```

Compare com a versão sequencial (basta comentar parallel()):

```less
Filter:a
Map:a
aa
Filter:b
Map:b
bb
Filter:c
Map:c
cc
Filter:d
Filter:e
Map:e
ee
1.554562 milliseconds
```

Neste caso, a versão sequencial teve melhor desempenho.

Mas se temos uma operação independente ou sem estado, onde a ordem não importa, por exemplo, contar o número de números pares em um intervalo grande, a versão paralela terá melhor desempenho:

```java
double start = System.nanoTime();
long c = IntStream.rangeClosed(0, 1_000_000_000)
            .parallel()
            .filter(i -> i % 2 == 0)
            .count();
double duration = (System.nanoTime() - start) / 1_000_000;
System.out.println("Got " + c + " in " + duration + " milliseconds");
```

Saída da versão paralela:

```nginx
Got 500000001 in 738.678448 milliseconds
```

Saída da versão sequencial:

```yaml
Got 500000001 in 1275.271882 milliseconds
```

Em resumo

Streams paralelos nem sempre têm desempenho melhor que streams sequenciais.
Isso, o fato de que streams paralelos processam resultados de forma independente e que a ordem não pode ser garantida são os pontos mais importantes que você precisa saber.

Mas na prática, como saber quando usar streams sequenciais ou paralelos para obter melhor desempenho?

Aqui estão algumas regras:

- Para um conjunto pequeno de dados, streams sequenciais são quase sempre a melhor escolha devido à sobrecarga do paralelismo.
- Ao usar streams paralelos, evite operações com estado (como sorted()) e baseadas em ordem (como findFirst()).
- Operações que são computacionalmente caras (considerando toda a operação no pipeline), geralmente têm melhor desempenho usando um stream paralelo.
- Em caso de dúvida, verifique o desempenho com um benchmark apropriado.

Reduzindo Streams Paralelos

Em ambientes concorrentes, atribuições são ruins.

Isso porque, se você mutar o estado de variáveis (especialmente se elas forem compartilhadas por mais de uma thread), poderá ter muitos problemas para evitar estados inválidos.

Considere este exemplo, que implementa o fatorial de 10 de um modo bastante particular:

```java
class Total {
    public long total = 1;
    public void multiply(long n) { total *= n; }
}
...
Total t = new Total();
LongStream.rangeClosed(1, 10)
    .forEach(t::multiply);
System.out.println(t.total);
```

Aqui, estamos usando uma variável para reunir o resultado do fatorial. A saída ao executar este trecho de código é:

```
3628800
```

Contudo, ao tornarmos o stream paralelo:

```java
LongStream.rangeClosed(1, 10)
    .parallel()
    .forEach(t::multiply);
```

Às vezes obtemos o resultado correto, e outras vezes não.

O problema é causado pelas múltiplas threads acessando a variável total simultaneamente. Sim, podemos sincronizar o acesso a essa variável (como veremos em um capítulo posterior), mas isso meio que anula o propósito do paralelismo (eu disse que atribuições são ruins em ambientes concorrentes).

É aí que reduce() é útil.

Se você se lembra do capítulo anterior, reduce() combina os elementos de um stream em um único.

Com streams paralelos, este método cria valores intermediários e depois os combina, evitando o problema de "ordem" ao mesmo tempo que permite que os streams sejam processados em paralelo ao eliminar o estado compartilhado e mantê-lo dentro do processo de redução.

O único requisito é que a operação de redução aplicada deve ser associativa.

Isso significa que a operação op deve seguir esta igualdade:

```
r                                                    Copiar  Editar

(a op b) op c == a op (b op c)
```

Ou:

```
r                                                    Copiar  Editar

a op b op c op d == (a op b) op (c op d)
```

Assim podemos avaliar (a op b) e (c op d) em paralelo.

Voltando ao nosso exemplo, podemos implementá-lo usando parallel() e reduce() desta maneira:

```
java                                                 Copiar  Editar

long tot = LongStream.rangeClosed(1, 10)
                .parallel()
                .reduce(1, (a,b) -> a*b);
System.out.println(tot);
```

Ao executar este trecho de código, ele produz o resultado correto toda vez (3628800).

E se cronometrarmos a execução do primeiro trecho (47.216181 milissegundos) e deste último (3.094717 milissegundos), podemos ver uma grande melhoria de desempenho. Claro, esses valores (e os outros apresentados neste capítulo) dependem da potência da máquina, mas você deverá obter resultados semelhantes.

Também podemos aplicar redução ao exemplo apresentado no início deste capítulo para processar a string em paralelo mantendo a ordem:

```
java                                                 Copiar  Editar

String s = Stream.of("a","b","c","d","e")
            .parallel()
            .reduce("", (s1, s2) -> s1 + s2);
System.out.println(s);
```

Saída:

```
nginx                                                Copiar  Editar

abcde
```

Estes são exemplos simples, mas redução é difícil de acertar às vezes, então é melhor evitar estado mutável compartilhado e usar operações sem estado e independentes para garantir que streams paralelos produzam os melhores resultados.

Por exemplo, lembra deste exemplo do capítulo anterior?

```java
int total = IntStream.of(1, 2, 3, 4, 5, 6)
            .reduce(4, (sum, n) -> sum + n);
```

Neste exemplo, o primeiro parâmetro não é realmente uma identidade, já que 4 + n não é igual a n.

Quando tornei o stream paralelo, em vez de obter o resultado esperado (25), obtive 45:

```java
int total = IntStream.of(1, 2, 3, 4, 5, 6)
            .parallel()
            .reduce(4, (sum, n) -> sum + n);
```

Por quê?

Porque o stream é dividido em partes, e a função acumuladora é aplicada a cada uma de forma independente, o que significa que 4 é adicionado não apenas ao primeiro elemento, mas ao primeiro elemento de **cada** parte.

Redução com Três Argumentos

Como também vimos no capítulo anterior, a versão de reduce() que recebe três argumentos é particularmente útil em streams paralelos.

Por exemplo, se pegarmos o exemplo do capítulo anterior que soma o comprimento de algumas strings e o tornarmos paralelo:

```java
int length = Stream.of("Parallel", "streams", "are", "great")
             .parallel()
             .reduce(0,
                (accumInt, str) ->
                   accumInt + str.length(), // acumulador
                (accumInt1, accumInt2) ->
                   accumInt1 + accumInt2); // combinador
```

Podemos ver o que acontece para obter o resultado (23):

1. A função acumuladora é aplicada a cada elemento sem uma ordem específica. Por exemplo:

```cpp
0 + 5 // great
0 + 3 // are
0 + 8 // Parallel
0 + 7 // streams
```

Os resultados da função acumuladora são combinados:

```
5 + 3 = 8
8 + 7 = 15
8 + 15 = 23
```

Quando estamos retornando um valor reduzido do tipo U a partir de elementos do tipo T, e em um stream paralelo os elementos são divididos em N resultados intermediários do tipo U, faz sentido ter uma função que saiba como combinar esses valores U em um único resultado.

Por isso, se não estivermos usando tipos diferentes, a função acumuladora **é** a mesma que a função combinadora.

Finalmente

Assim como reduce(), podemos usar com segurança collect() com streams paralelos se seguirmos os mesmos requisitos de **associatividade** e **identidade**, como, por exemplo: para qualquer resultado parcialmente acumulado, combiná-lo com um recipiente de resultado vazio deve produzir um resultado equivalente.

Ou, se estivermos agrupando com a classe Collectors e a ordenação não for importante, podemos usar o método groupingByConcurrent(), a versão concorrente de groupingBy().

Pontos-Chave

- Streams paralelos dividem o stream em múltiplas partes. Cada parte é processada por uma thread diferente ao mesmo tempo (em paralelo).
- Para criar um stream paralelo a partir de outro stream, use o método parallel().
- Para criar um stream paralelo a partir de uma Collection, use o método parallelStream().
- Streams paralelos são mais apropriados para operações onde a ordem de processamento não importa e que não precisam manter estado (elas são sem estado e independentes).
- Você pode transformar um stream paralelo em sequencial com o método sequential().
- Você pode verificar se um stream é paralelo com isParallel().
- Você pode tornar um stream ordenado não ordenado (ou garantir que o stream seja não ordenado) com unordered().
- Streams paralelos **nem sempre** têm desempenho melhor do que streams sequenciais.
- Com streams paralelos, reduce() cria valores intermediários e depois os combina, evitando o problema de "ordem" e ainda permitindo que os streams sejam processados em paralelo ao eliminar o estado compartilhado (mutável) e mantê-lo dentro do processo de redução.
- O único requisito é que a operação de redução aplicada seja associativa:

```
r
(a op b) op c == a op (b op c)
```

Autoavaliação

1. Dado:

```java
public class Question_18_1 {
    public static void main(String[] args) {
        OptionalInt sum = IntStream.rangeClosed(1, 10)
                           .parallel()
                           .unordered()
                           .reduce(Integer::sum);
        System.out.println(sum.orElse(0));
    }
}
```

Qual é o resultado?

A. 0
B. 55 é impresso o tempo todo
C. Às vezes 55 é impresso
D. Uma exceção ocorre em tempo de execução

---

2. Qual das afirmações a seguir é verdadeira?

A. Você pode chamar o método parallel() em uma Collection para criar um stream paralelo.
B. Operações que são computacionalmente caras geralmente têm melhor desempenho usando um stream sequencial.
C. filter() é um método sem estado.
D. Streams paralelos sempre têm desempenho melhor que streams sequenciais.

---

3. Dado:

```java
public class Question_18_3 {
    public static void main(String[] args) {
        OptionalDouble avg =
                IntStream.rangeClosed(1, 10)
                           .parallel()
                           .average();
        System.out.println(avg.getAsDouble());
    }
}
```

Qual é o resultado?

A. 5.5 é impresso o tempo todo
B. Às vezes 5.5 é impresso
C. Falha na compilação
D. Uma exceção ocorre em tempo de execução

---

4. Dado:

```java
public class Question_17_4 {
    public static void main(String[] args) {
        IntStream.of(1, 1, 3, 3, 7, 6, 7)
            .distinct()
            .parallel()
            .map(i -> i*2)
            .sequential()
            .forEach(System.out::print);
    }
}
```

Qual é o resultado?

A. Pode imprimir 142612 às vezes
B. 1133767
C. Pode imprimir 3131677 às vezes
D. 261412